Open in app ↗

Search

Rate Limiter

Suresh Podeti
9 min read · Sep 26, 2023

Listen Share More

Photo by Yassine Khalfalli on Unsplash

Introduction

A **rate limiter**, puts a limit on the number of requests a service fulfills. It throttles requests that cross the predefined limit. For example, a client using a particular service's API that is configured to allow 500 requests per minute would block further incoming requests for the client if the number of requests the client makes exceeds that limit.

A rate limiter is generally used as a defensive layer for services to avoid their excessive usage, whether intended or unintended. It also protects services against abusive behaviours that target the application layer, such as **denial-of-service** (**DOS**) attacks and brute-force password attempts, preventing resource starvation, and managing policies and quotas.

Design

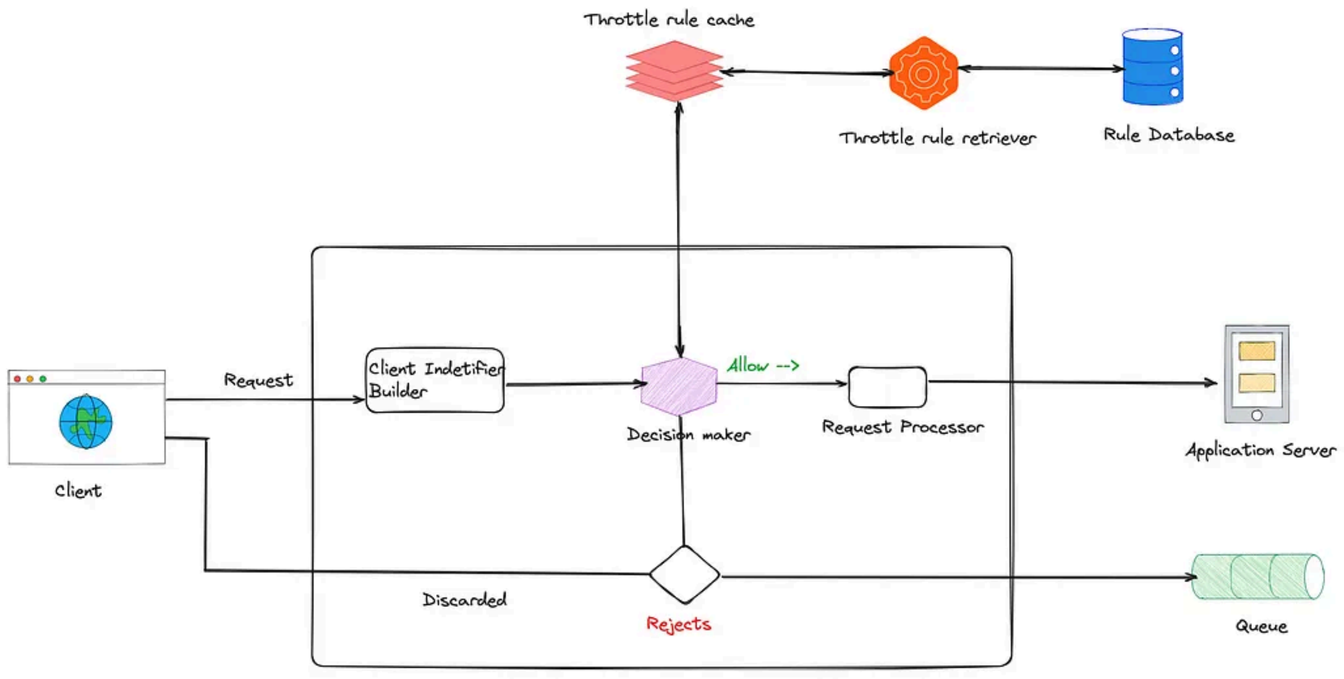

Fig 1.0: The rate limiter accepts or rejects requests based on throttle rules

Let's discuss each component that is present in the detailed design of a rate limiter.

Rule database: This is the database, consisting of rules defined by the service owner. Each rule specifies the number of requests allowed for a particular client per unit of time.

Rules retriever: This is a background process that periodically checks for any modifications to the rules in the database. The rule cache is updated if there are any modifications made to the existing rules.

Throttle rules cache: The cache consists of rules retrieved from the **rule database**. The cache serves a rate-limiter request faster than persistent storage. As a result, it increases the performance of the system. So, when the rate limiter receives a request against an ID (key), it checks the ID against the rules in the cache. It can be used for storing the data required for computing rate limiting algorithms.

Decision-maker: This component is responsible for making decisions against the rules in the cache. This component works based on one of the rate-limiting algorithms that are discussed in the next lesson.

Client identifier builder: This component generates a unique ID for a request received from a client. This could be a remote IP address, login ID, or a combination of several other attributes, due to which a sequencer can't be used here. This ID is considered as a key to store the user data in the key-value database. So, this key is passed to the **decision-maker** for further service decisions.

In case the predefined limit is crossed, APIs return an HTTP response code `429 Too Many Requests` , and one of the following strategies is applied to the request:

- Drop the request and return a specific response to the client, such as "too many requests" or "service unavailable."
- If some requests are rate limited due to a system overload, we can keep those requests in a queue to be processed later.

Request processing

When a request is received, the *client identifier builder* identifies the request and forwards it to the *decision-maker*. The decision-maker determines the services required by request, then checks the cache against the number of requests allowed, as well as the rules provided by the service owner. If the request does not exceed the count limit, it is forwarded to the *request processor*, which is responsible for serving the request.

The decision-maker takes decisions based on the throttling algorithms. The throttling can be hard, soft, or elastic. Based on **soft or elastic throttling**, requests are allowed more than the defined limit. These requests are either served or kept in the queue and served later, upon the availability of resources. Similarly, if **hard throttling** is used, requests are rejected, and a response error is sent back to the client.

Race condition

There is a possibility of a race condition in a situation of high concurrency request patterns. It happens when the "**get-then-set**" approach is followed, wherein the current counter is retrieved, incremented, and then pushed back to the database. While following this approach, some additional requests can come through that could leave the **incremented counter invalid**. This allows a client to send a very high rate of requests, bypassing the rate-limiting controls. To avoid this problem, the **locking mechanism** can be used, where one process can update the counter at a time while others wait for the lock to be released. Since this approach can cause a **potential bottleneck**, it significantly **degrades performance** and does not scale well.

Another method that could be used is the "**set-then-get**" approach, wherein a value is incremented in a very performant fashion, avoiding the locking approach. This approach works if there's minimum contention.

A rate limiter should not be on the client's critical path

Let's assume a real-time scenario where millions of requests hit the front-end servers. Each request will **retrieve, update, and push back the count** to the respective cache. After all these operations, the request is sent forward to be served. This approach could cause latency if there is a high number of requests. To **avoid numerous computations in the client's critical path**, we should **divide the work into offline and online parts**.

Initially, when a client's request is received, the system will just check the respective count. If it is less than the maximum limit, the system will allow the client's request. In the second phase, the system updates the respective count and cache offline. For a few requests, this won't have any effect on the performance, but for millions of requests, this approach increases performance significantly.

Algorithms

Token bucket algorithm

This algorithm uses the analogy of a **bucket** with a **predefined capacity of tokens**. The bucket is periodically **filled** with tokens **at a constant rate**. A token can be considered as a **packet of some specific size**. Hence, the algorithm checks for the token in the bucket each time we receive a request. There should be at least one token to process the request further.

Assume that we have a predefined rate limit of R and the total capacity of the bucket is C:

- The algorithm adds a new token to the bucket after every $1/R$ seconds.

- The algorithm discards the new incoming tokens when the number of tokens in the bucket is equal to the total capacity *C* of the bucket.
- If there are *N* incoming requests and the bucket has at least *N* tokens, the tokens are consumed, and requests are forwarded for further processing.
- If there are *N* incoming requests and the bucket has a lower number of tokens, then the number of requests accepted equals the number of available tokens in the bucket.

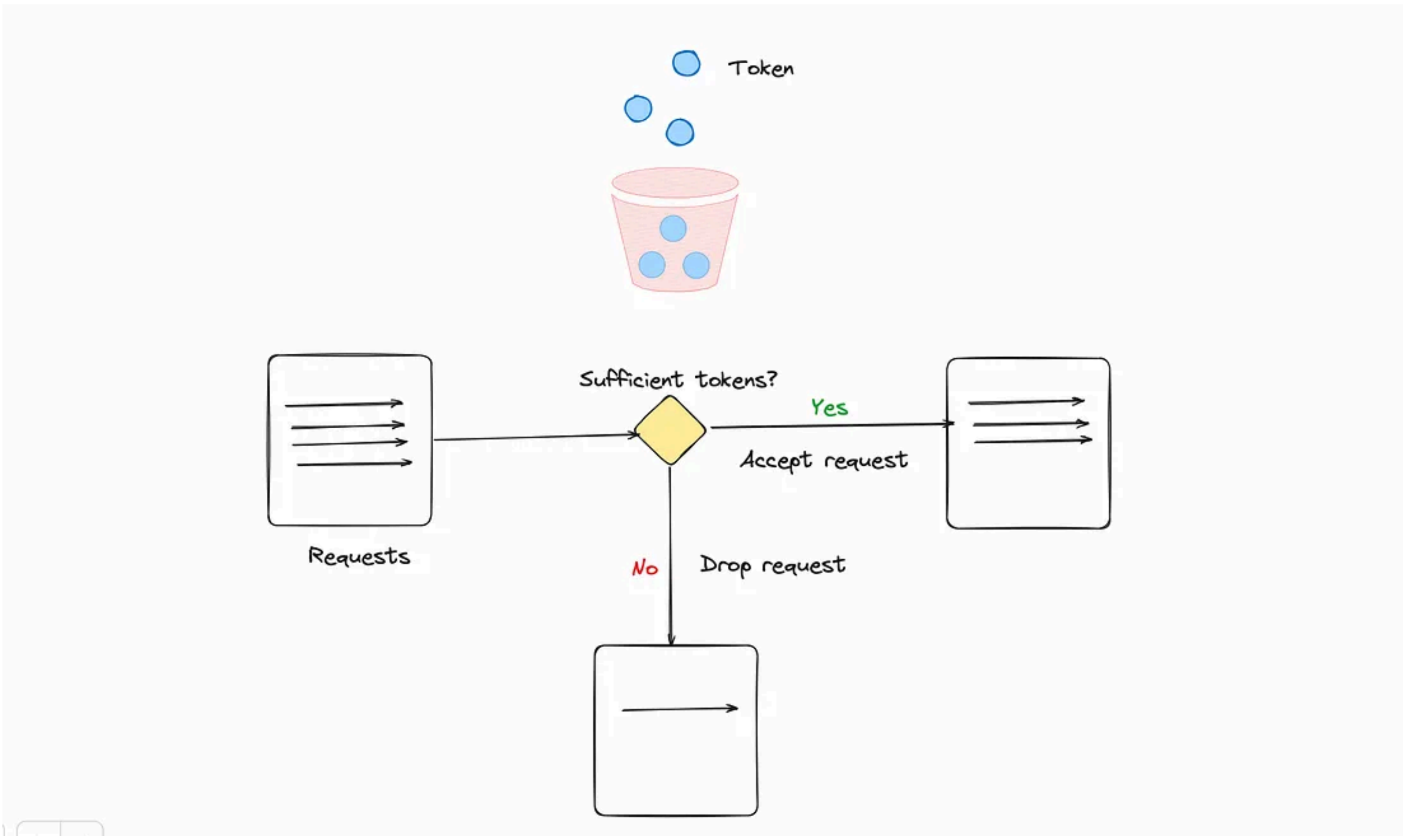

Fig 2.0: Token bucket algorithm

Advantages

- This algorithm can cause a burst of traffic as long as there are enough tokens in the bucket.
- It is space efficient. The memory needed for the algorithm is nominal due to limited states.

Disadvantages

- From the implementation perspective, a **lock** might require taking a token from the bucket that can increase the request's processing delay if **contention on the lock increases.**
- Choosing an optimal value for the essential parameters is a difficult task.

The leaking bucket algorithm

The **leaking bucket algorithm** is a variant of the token bucket algorithm with slight modifications. Instead of using tokens, the leaking bucket algorithm uses a **bucket to contain incoming requests** and processes them at a constant outgoing rate. This algorithm uses the analogy of a water bucket leaking at a constant rate. Similarly, in this algorithm, the requests arrive at a variable rate. The algorithm process these requests at a constant rate in a first-in-first-out (FIFO) order.

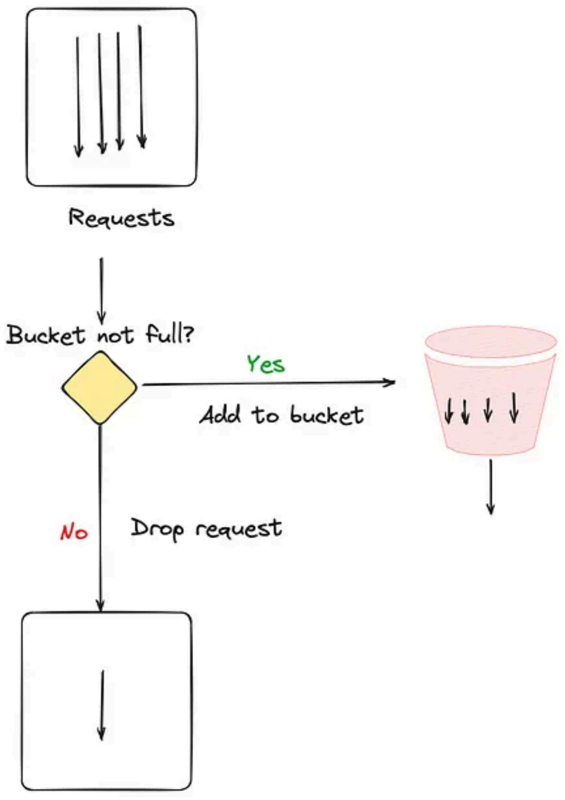

Fig 3.0: Leaking bucket algorithm

Advantages

- Due to a constant outflow rate, it avoids the burst of requests, unlike the token bucket algorithm.
- This algorithm is also space efficient.

- Since requests are processed at a fixed rate, it is suitable for **applications with a stable outflow rate**.

Disadvantages

- A burst of requests can fill the bucket, and if not processed in the specified time, recent requests can take a hit.
- Determining an optimal bucket size and outflow rate is a challenge.

Fixed window counter algorithm

This algorithm divides the time into **fixed intervals** called **windows** and assigns a counter to each window. When a specific window receives a request, the **counter is incremented by one**. Once the counter reaches its limit, new requests are discarded in that window.

As shown in the below figure, a dotted line represents the limit in each window. If the counter is lower than the limit, forward the request; otherwise, discard the request.

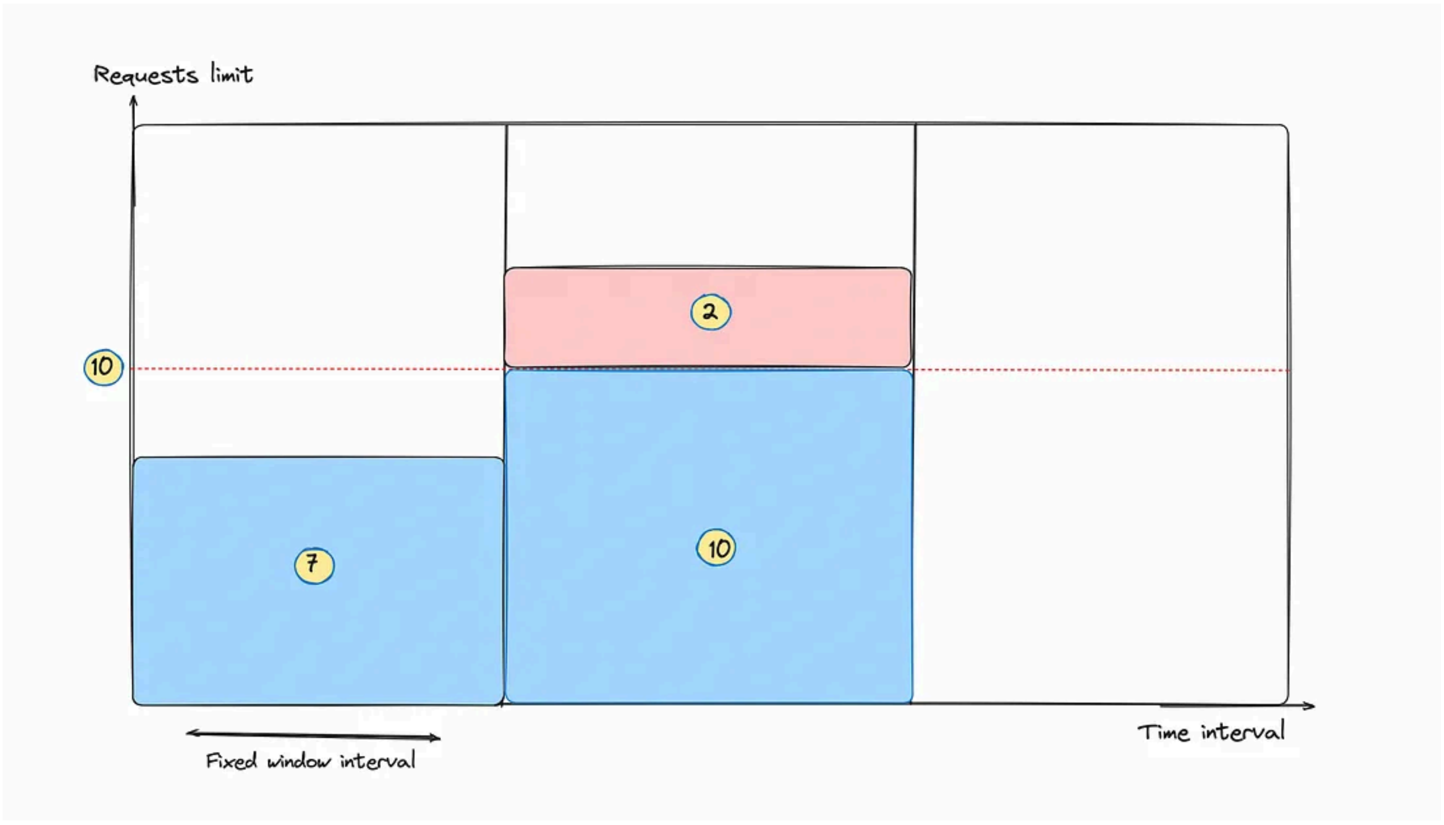

Fig 3.0: Fixed window counter algorithm: Discard the request exceeding the limit

Advantages

- It is also space efficient due to constraints on the rate of requests.
- As compared to token bucket-style algorithms (that discard the new requests if there aren't enough tokens), this algorithm services the new requests.

Disadvantages

- A consistent burst of traffic (twice the number of allowed requests per window) at the window edges could cause a potential decrease in performance — A burst of traffic greater than the allowed requests can occur at the edges of the window

Sliding window log algorithm

The **sliding window log algorithm** keeps track of each incoming request. When a request arrives, its arrival time is stored in a hash map, usually known as the log. The logs are sorted based on the time stamps of incoming requests. **The requests are allowed depending on the size of the log and arrival time**.

The main advantage of this algorithm is that it doesn't suffer from the edge conditions, as compared to the **fixed window counter** algorithm.

Advantages

- The algorithm doesn't suffer from the boundary conditions of fixed windows.

Disadvantages

- It **consumes extra memory** for storing additional information, the time stamps of incoming requests. It keeps the time stamps to provide a dynamic window, even if the request is rejected.

Sliding window counter algorithm

Fixed window counter + Sliding window log → Sliding window counter

Unlike the previously fixed window algorithm, the **sliding window counter algorithm** doesn't limit the requests based on fixed time units. This algorithm takes into account both the fixed window counter and sliding window log algorithms to make the flow of requests more smooth. Let's look at the flow of the algorithm in the below figure.

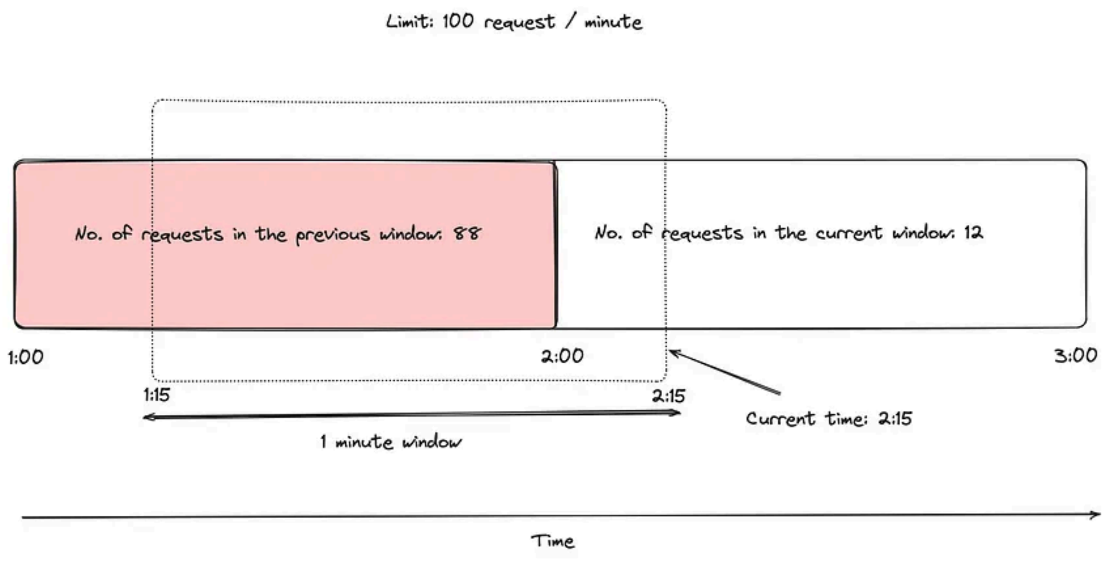
Limit: 100 request / minute
No. of requests in the previous window: 88
No. of requests in the current window: 12
1:00
2:00
3:00
1:15
2:15
Current time: 2:15
1 minute window
Time

Fig 4.0: A sliding window counter algorithm, where the green shaded area shows the rolling window of 1 minute

In the above figure, we've 88 requests in the previous window while 12 in the current window. We've set the rate limit to 100 requests per minute. Further, the rolling window overlaps 15 seconds with the current window. Now assume that a new request arrives at 02:15. We'll decide which request to accept or reject using the mathematical formulation:

$Rate = Rp \times (time\ frame - overlap\ time) / time\ frame + Rc$

Here, *Rp* is the **number of requests in the previous window**, which is 88. *Rc* is the number of requests in the current window, which is 12. The *time frame* is 60 seconds in our case, and *overlap time* is 15 seconds.

$Rate = 88 \times (60 - 15)/60 + 12$

$Rate = 78 < 100$

As 78 is less than 100, so the incoming request is allowed.

Advantages

- The algorithm is also space efficient due to limited states: the number of requests in the current window, the number of requests in the previous window, the overlapping percentage, and so on.

- It **smooths out the bursts** of requests and processes them with an approximate average rate based on the previous window.

Disadvantages

- This algorithm assumes that the number of requests in the previous window is evenly distributed, which **may not always be possible**.

Evaluation

- **Availability:** If a rate limiter fails, multiple rate limiters will be available to handle the incoming requests. So, a single point of failure is eliminated.
- **Low latency:** Our system retrieves and updates the data of each incoming request from the cache instead of the database. First, the incoming requests are forwarded if they do not exceed the rate limit, and then the cache and database are updated.
- **Scalability:** The number of rate limiters can be increased or decreased based on the number of incoming requests within the defined limit.

Rate Limiting System Design Interview Distributed Systems Software Architecture Software Architect

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience